Appunti di semantica

Programmazione I e Laboratorio
(corsi A-B)

Corso di Laurea in Informatica
Università di Pisa
A.A. 2011/12

R. Barbuti, P. Mancarella

Indice

Gli studenti sono pregati di consultare questi appunti con spirito critico e di segnalare tempestivamente errori o refusi ai docenti.

1 Sintassi

Introduciamo la sintassi del frammento del linguaggio `C` utilizzato nel corso.

1.1 Espressioni

```
Exp ::= Num | Ide | Exp Aop Exp | Exp Bop Exp | Uop Exp | (Exp)

Aop ::= + | - | * | / | %
Bop ::= > | >= | < | <= | == | != | && | ||
Uop ::= !

Num  ::= Digit |  Num Digit
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

1.2 Dichiarazioni

```
Dec ::= Type Ide; | Type Ide = Exp;

Type ::= BType

BType ::= int
```

1.3 Comandi

```
Com ::= Ide = Exp; | if (Exp) Com else Com | if (Exp) Com | while (Exp) Com | Block

Block ::= {DecComList}

DecComList ::= DecList ComList | ComList

DecList ::= Dec | Dec DecList
ComList ::= Com | Com ComList
```

Programmi

```
Prog ::= Block
```

1.3.1 Rappresentazione in CAML della sintassi

```
type ide == string;;


(* ESPRESSIONI *)
type bop = Add | Sub | Mul | Div | Mod |
           Gt | Gte | Lt | Lte | Eq | Neq | And | Or;;

type uop = Minus | Not;;

type exp = Ide of ide |
           Num of int |
           Bool of bool |
           BinExp of exp*bop*exp |
           UnExp of uop*exp;;

(* DICHIARAZIONI *)

type dec = Var of ide |
           Var_init of ide * exp;;

(* COMANDI *)

type com = Assign of ide * exp |
           If_then_else of exp*com*com |
           If_then of exp*com |
           While of exp*com |
           Block of dec list * com list |
           Call of ide*exp;;

type prog = Prog of dec list * com list;;
```

2 Lo stato

Domini semantici di base

- `Ide` è il dominio degli identificatori
- `Val` è il dominio dei valori
- `Loc` è il dominio delle locazioni di memoria

Dato un dominio semantico D, indichiamo con $D_\perp$ il dominio $D \cup \{\perp\}$, dove si assume $\perp \notin D$.

2.1 Frame e pile

Siano A e B due insiemi. Un *frame* f è una funzione parziale da A in B, ovvero una funzione totale da A in $B_\perp$.

$$f : A \to B_\perp$$

Con ω denoteremo in seguito il frame tale che $\omega(x) = \perp$, per ogni $x \in A$.

Se è finito l'insieme degli elementi di A sul quale un frame f ha valore diverso da $\perp$, è spesso utile darne una rappresentazione tabellare. Ad esempio, la tabella

x	25
y	10
z	−12
w	38

rappresenta il seguente frame $f : \texttt{Ide} \to \texttt{Val}_\perp$

$$f(id) = \begin{cases} 25 & \text{se } id = x \\ 10 & \text{se } id = y \\ -12 & \text{se } id = z \\ 38 & \text{se } id = w \\ \perp & \text{altrimenti} \end{cases}$$

Dati A e B, sia $\mathcal{F}$ l'insieme dei frame $f : A \to B_\perp$. Una *pila di frame* è allora un elemento dell'insieme Π così definito in modo ricorsivo:

$$\Pi = \big\{\Omega\big\} \cup \big\{f.\pi \mid f \in \mathcal{F}, \pi \in \Pi\big\}$$

Nel seguito diremo che Ω è la pila vuota.

Anche delle pile è spesso utile una rappresentazione grafica tabellare. Consideriamo ad esempio la pila $\pi = f1.f2.\Omega$ dove $f1$ ed $f2$ sono i seguenti frame nella loro rappresentazione tabellare:

x	10
y	30

f1

z	25
y	2

f2

La rappresentazione di π che useremo è la seguente

x	10
y	30

z	25
y	2

Si noti che la pila precedente è strutturalmente diversa dalla pila $f2.f1.\Omega$, la cui rappresentazione grafica è:

z	25
y	2

x	10
y	30

2.2 Operazioni su frame e su pile

Sui frame e sulle pile sono definite le operazioni di *ricerca* di una associazione, *aggiunta* di una associazione di *modifica* di una associazione. Sia

$$f : A \rightarrow B_{\perp}$$

un frame e siano $a \in A$ e $b \in B$.

Operazioni su frame

Ricerca *Ricerca di una associazione*
Il valore associato ad un elemento $a \in A$ è semplicemente $f(a)$. Si noti che $f(a) = \perp$ denota l'assenza di una associazione per a.

Add *Aggiunta di una nuova associazione*
Nell'ipotesi $f(a) = \perp$, con $f[^b/_a]^{add}$ indichiamo un nuovo frame $g : A \rightarrow B_{\perp}$ così definito:

$$g(x) = \begin{cases} b & \text{se } x = a \\ f(x) & \text{altrimenti} \end{cases}$$

Nella rappresentazione tabellare, il nuovo frame g ha una riga in più (la riga per a) rispetto a f. Dato ad esempio il frame f rappresentato dalla seguente tabella

x	10
y	30

il frame $f[^{50}/_{z}]^{add}$ è rappresentato dalla tabella

x	10
y	30
z	50

mentre non è definito, ad esempio, il frame $f[^{50}/_{x}]^{add}$, dal momento che $f(x) \neq \bot$.

Mod *Modifica di una associazione esistente*
Nell'ipotesi $f(a) \neq \bot$, con $f[^{b}/_{a}]^{mod}$ indichiamo un nuovo frame $g : A \to B_\bot$ così definito:

$$g(x) = \begin{cases} b & \text{se } x = a \\ f(x) & \text{altrimenti} \end{cases}$$

Nella rappresentazione tabellare, il nuovo frame g si differenzia da f nella riga per a. Dato ad esempio il frame f rappresentato dalla seguente tabella

x	10
y	30

il frame $f[^{50}/_{x}]^{mod}$ è rappresentato dalla tabella

x	50
y	30

mentre non è definito, ad esempio, il frame $f[^{50}/_{z}]^{mod}$, dal momento che $f(z) = \bot$.

Notazioni

Sia $f : A \to B_\bot$ un frame e siano $a_1, a_2 \in A$ e $b_1, b_2 \in B$. In seguito useremo l'abbreviazione

$$f[^{b_1}/_{a_1}, ^{b_2}/_{a_2}]^{add}$$

per rappresentare il frame

$$\left(f[^{b_1}/_{a_1}]^{add}\right)[^{b_2}/_{a_2}]^{add}$$

in cui necessariamente a_1 ed a_2 sono distinti tra loro.

Analogamente, nell'ipotesi che a_1 ed a_2 siano *distinti*, useremo l'abbreviazione

$$f[^{b_1}/_{a_1}, ^{b_2}/_{a_2}]^{mod}$$

per rappresentare il frame

$$\left(f[^{b_1}/_{a_1}]^{mod}\right)[^{b_2}/_{a_2}]^{mod}.$$

Si faccia attenzione al fatto che una tale abbreviazione non avrebbe senso nel caso in cui a_1 ed a_2 siano lo stesso elemento $a \in A$. In questo caso, possiamo invece ragionare sulla definizione di $[\ /\ ]^{mod}$ al fine di ottenere comunque una scrittura più compatta per

$$\big(f[^{b_1}/_a]^{mod}\big)[^{b_2}/_a]^{mod}$$

Abbiamo infatti:

$$\big(f[^{b_1}/_a]^{mod}\big)[^{b_2}/_a]^{mod}(x) = \begin{cases} b_2 & \text{se } x = a \\ \big(f[^{b_1}/_a]^{mod}(x) & \text{se } x \neq a \end{cases}$$

Ma per definizione di $[\ /\ ]^{mod}$, è facile osservare che, se $x \neq a$

$$\big(f[^{b_1}/_a]^{mod}\big)(x) = f(x)$$

e dunque otteniamo

$$\big(f[^{b_1}/_a]^{mod}\big)[^{b_2}/_a]^{mod}(x) = \begin{cases} b_2 & \text{se } x = a \\ f(x) & \text{se } x \neq a \end{cases}$$

che ci autorizza a scrivere

$$\big(f[^{b_1}/_a]^{mod}\big)[^{b_2}/_a]^{mod} = f[^{b_2}/_a]^{mod}$$

Operazioni su pile

Dati A e B, sia $\mathcal{F}$ l'insieme dei frame $f : A \to B_\perp$ e sia Π l'insieme delle pile di frame in $\mathcal{F}$:

$$\Pi = \big\{\Omega\big\} \cup \big\{f.\pi \mid f \in \mathcal{F}, \pi \in \Pi\big\}$$

Ricerca *Ricerca di una associazione*

Sia $\pi \in \Pi$ una pila del tipo $f_1.f_2.\ldots.f_n.\Omega$. Il valore $\pi(a)$ associato in π ad un elemento $a \in A$ è il valore associato ad a nel primo frame (da sinistra a destra) che contiene una associazione per a, o $\perp$ se nessuno tra i frame f_i, con $i = 1, \ldots, n$ contiene una associazione per a. Formalmente:

$$\pi(a) = \begin{cases} \perp & \text{se } \pi = \Omega \\ F(a) & \text{se } \pi = f.\pi' \wedge f(a) \neq \perp \\ \pi'(a) & \text{se } \pi = f.\pi' \wedge f(a) = \perp \end{cases}$$

Add *Aggiunta di una nuova associazione*

L'aggiunta di una associazione in una pila avviene sempre nel primo frame della pila. L'operazione è dunque definita solo su pile non vuote.

Siano $\pi \in \Pi$, $a \in A$ e $b \in B$. Con $\pi[^b/_a]^{add}$ indichiamo la nuova pila così definita

$$\pi[^b/_a]^{add} = f[^b/_a]^{add}.\pi' \qquad \text{se } \pi = f.\pi'$$

Mod *Modifica di una associazione esistente*

La modifica dell'associazione per un elemento $a \in A$ in una pila avviene sempre nel primo frame della pila che contiene una associazione per a. L'operazione è dunque definita solo su pile che contengono una associazione per a, ovvero per cui $\pi(a) \neq \perp$.

Siano $\pi \in \Pi$, $a \in A$ e $b \in B$. Con $\pi[^b/_a]^{mod}$ indichiamo la nuova pila così definita

$$\pi[^b/_a]^{mod} = \begin{cases} f[^b/_a]^{mod}.\pi' & \text{se } \pi = f.\pi' \wedge f(a) \neq \bot \\ f.\pi'[^b/_a]^{mod} & \text{se } \pi = f.\pi' \wedge f(a) = \bot \end{cases}$$

Si noti che la modifica $\pi[^b/_a]^{mod}$ non è definita su una pila che non contiene una associazione per a (perché?).

2.2.1 Implementazione in CAML di frame e pile

```
(* FRAME E STACK *)
type 'a bottom = Bottom | Def of 'a;;

let omega x = Bottom;;


let add f x y =
        match f x with
          Bottom -> let g z = if z=x then y else f z in g;;

let update f x y =
        match f x with
          z when z<>Bottom -> let g z = if z=x then y else f z in g;;


let rec add_stack pi x y =
        match pi with
          f::pi' -> (add f x y)::pi';;


let rec update_stack pi x y =
        match pi with
          f::pi' when f x <> Bottom -> (update f x y)::pi' |
          f::pi' when f x = Bottom -> f :: (update_stack pi' x y);;

let rec search pi x =
        match pi with
          [] -> Bottom |
          f :: pi' when f x <> Bottom -> f x |
          f :: pi' when f x = Bottom -> search pi' x;;
```

Tipizzazione

```
omega : 'a -> 'b bottom
add : ('a -> 'b bottom) -> 'a -> 'b bottom -> 'a -> 'b bottom
update : ('a -> 'b bottom) -> 'a -> 'b bottom -> 'a -> 'b bottom
add_stack : ('a -> 'b bottom) list -> 'a -> 'b bottom -> ('a -> 'b bottom) list
update_stack : ('a -> 'b bottom) list -> 'a -> 'b bottom -> ('a -> 'b bottom) list
```

```
search : ('a -> 'b bottom) list -> 'a -> 'b bottom
```

2.3 Ambiente e Memoria

Uno stato è costituito da una coppia di pile, detti rispettivamente *ambiente* e *memoria*. Un ambiente è una pila di *frame ambiente*. Una memoria è una pila di *frame memoria*.

- Un *frame ambiente* è una funzione

  $\varphi : \texttt{Ide} \to \texttt{Loc}_\perp$

  L'insieme dei frame ambiente viene denotato con Φ ed è così definito:

  $$\Phi = \{\varphi \mid \varphi : \texttt{Ide} \to \texttt{Loc}_\perp\}$$

- Un *frame memoria* è una funzione

  $\nu : \texttt{Loc} \to \texttt{Val}_\perp$

  L'insieme dei frame memoria viene denotato con N ed è così definito:

  $$N = \{\nu \mid \nu : \texttt{Loc} \to \texttt{Val}_\perp\}$$

  Ricordiamo che con ω indichiamo sia il frame ambiente *vuoto* che il frame memoria *vuoto*. Detto altrimenti:

  per ogni x $\in$ `Ide` $\quad \omega(\text{x}) = \perp$ (frame ambiente vuoto)

  per ogni $\ell \in$ `Loc` $\quad \omega(\ell) = \perp$ (frame memoria vuoto)

- Un **ambiente** ρ è una *pila* (stack) di frame ambiente, ovvero un elemento dell'insieme P degli ambienti così definito:

  $$P = \{\Omega\} \cup \{\varphi.\rho \mid \varphi \in \Phi, \rho \in P\}$$

- Una **memoria** μ è una *pila* (stack) di frame memoria, ovvero un elemento dell'insieme M delle memorie così definito:

  $$M = \{\Omega\} \cup \{\nu.\mu \mid \nu \in N, \mu \in M\}$$

Operazioni sullo stato

Le operazioni di ricerca, aggiunta e modifica viste in precedenza su pile generiche sono definite anche sulle pile ambiente e memoria, ad eccezione dell'operazione di *modifica* che non è definita sulla pila ambiente, evidenziando così che è la memoria la sola componente modificabile dello stato. Riassumendo

	Ambiente	Memoria
Ricerca	SI	SI
Modifica	NO	SI
Aggiunta	SI	SI

2.3.1 Rappresentazione in CAML dello stato

```
(* VALORI *)
type val = ValN of int |
           ValB of bool |
           Unknown;;

(* LOCAZIONI *)
type mloc == int;;

(* Frame ambiente *)
type amb == ide -> mloc bottom;;

(* Frame memoria *)
type mem == mloc -> val bottom;;
```

3 Sintassi e semantica: i numeri naturali

Diamo in questo paragrafo un primo esempio di cosa significhi dare semantica, in stile denotazionale, ad un linguaggio. A tale scopo, consideriamo il linguaggio associato alla categoria sintattica `Num` introdotta in precedenza, che riportiamo di seguito per semplicità:

```
Num   ::= Digit |  Num Digit
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Vogliamo introdurre una *funzione di valutazione* che, data una stringa `n` nel linguaggio `Num` (scriveremo $\texttt{n} \in \texttt{Num}$ per brevità), ci dica quale è il numero naturale *rappresentato* dalla stringa, ovvero quale è il suo significato.

Per fare ciò dobbiamo innanzitutto introdurre la distinzione tra un numero naturale e la sua rappresentazione. Sia allora $\mathbb{N} = \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \ldots\}$ l'insieme dei numeri naturali, sul quale sono definite le usuali operazioni di somma $(+)$, prodotto $(\times)$, quoziente e resto della divisione intera (*div* e *mod*): assumeremo anche di avere le operazioni di "uguaglianza" $(=)$, "disuguaglianza" $(\neq)$, "minore"$(\prec)$, "maggiore" $(\succ)$, "maggiore o uguale" $(\succeq)$, "minore o uguale" $(\preceq)$ su coppie di valori in $\mathbb{N}$[1]. Ad ogni elemento `n` di `Num` corrisponde il *valore* in $\mathbb{N}$ di cui `n` è la *rappresentazione.* Tale associazione è data dalla seguente *funzione di valutazione*

$$val : \texttt{Num} \to \mathbb{N}$$

definita per casi nel modo seguente:

[1]Si noti la necessità di distinguere tra i simboli utilizzati per denotare le operazioni su $\mathbb{N}$ ed i simboli utilizzati per rappresentare sintatticamente gli stessi. Così, ad esempio, `+` è il simbolo *sintattico* utilizzato per rappresentare l'operazione di somma $+$ tra numeri naturali, ovvero tra elementi di $\mathbb{N}$. Analogamente, $\times, -, div, mod, =, \neq, \prec, \succ$ $, \succeq, \preceq$ sono operazioni su coppie di naturali e non vanno confuse con i corrispondenti simboli sintattici `*`, `-`, `/`, `%`, `=`, `!=` , `>`, `<`, `>=`, `<=`.

$$
\begin{array}{rcl}
val(\texttt{0}) & = & \underline{0} \\
val(\texttt{1}) & = & \underline{1} \\
 & \dots & \\
val(\texttt{9}) & = & \underline{9} \\
val(\texttt{nc}) & = & (val(\texttt{n}) \ \times \ \underline{10}) \ + \ val(\texttt{c})
\end{array}
$$

Si noti come i casi nella definizione di *val* corrispondano proprio ai casi della definizione sintattica di `Num`. Analogamente, è possibile definire la *funzione di rappresentazione*

$$rapp : \mathbb{N} \to \texttt{Num}$$

definita come segue:

$$
\begin{array}{rcll}
rapp(\underline{0}) & = & \texttt{0} & \\
rapp(\underline{1}) & = & \texttt{1} & \\
 & \dots & & \\
rapp(\underline{9}) & = & \texttt{9} & \\
rapp(\underline{n}) & = & rapp(\underline{n}\ div\ \underline{10})rapp(\underline{n}\ mod\ \underline{10}) & \text{se } \underline{n} \succ \underline{9}
\end{array}
$$

La scelta di denotare i *valori* in $\mathbb{N}$ con la notazione $\underline{\texttt{n}}$ non deve indurre a pensare che la funzione di valutazione (risp. rappresentazione) possa essere definita semplicemente come $val(\texttt{n}) = \underline{\texttt{n}}$ (risp. $rapp(\underline{\texttt{n}}) = \texttt{n}$). La notazione $\underline{\texttt{n}}$ è anch'essa una *rappresentazione*, diversa da quella sintattica, da noi scelta per denotare un *valore* in $\mathbb{N}$. La funzione di valutazione definita per casi fornisce un modo per calcolare in modo *costruttivo* il valore corrispondente ad una stringa della particolare sintassi (linguaggio di rappresentazione) utilizzata (nel nostro caso, di una stringa di `Num`).

Per convincerci ulteriormente della necessità di distinguere tra *valori* e *rappresentazioni*, utilizziamo una sintassi diversa per la descrizione di numeri naturali, che corrisponde alla rappresentazione *binaria* dei numeri. Utilizziamo volutamente i simboli `z` e `u`, anziché gli usuali simboli `0` e `1`, per le cifre binarie (*bit*) proprio per evidenziarne il carattere puramente *simbolico*.

```
Binary ::= Bit | Binary Bit
Bit ::= z | u
```

Qual è il numero naturale rappresentato dalla stringa `zuzz`? Definiamo per casi una nuova funzione di valutazione

$$val' : \texttt{Binary} \to \mathbb{N}$$

per il nuovo linguaggio. Nella definizione di val' utilizziamo la convenzione di denotare con x una generica stringa in `Binary` e con b un generico bit (ovvero b sta per `z` oppure `u`).

$$
\begin{array}{rcl}
val'(\texttt{z}) & = & \underline{0} \\
val'(\texttt{u}) & = & \underline{1} \\
val'(xb) & = & (val'(x) \ \times \ \underline{2}) \ + \ val'(b)
\end{array}
$$

Applicando la funzione val' alla stringa `zuzz` otteniamo:

$$\begin{aligned}
val'(\texttt{zuzz}) &= (val'(\texttt{zuz}) \times \underline{2}) + val'(\texttt{z}) \\
&= (val'(\texttt{zuz}) \times \underline{2}) + \underline{0} \\
&= (((val'(\texttt{zu}) \times \underline{2}) + val'(\texttt{z})) \times \underline{2}) + \underline{0} \\
&= (((val'(\texttt{zu}) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\
&= (((((val'(\texttt{z}) \times \underline{2}) + val'(\texttt{u})) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\
&= (((((\underline{0} \times \underline{2}) + \underline{1}) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\
&= ((((\underline{0} + \underline{1}) \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\
&= (((\underline{1} \times \underline{2}) + \underline{0}) \times \underline{2}) + \underline{0} \\
&= ((\underline{2} + \underline{0}) \times \underline{2}) + \underline{0} \\
&= (\underline{2} \times \underline{2}) + \underline{0} \\
&= \underline{4} + \underline{0} \\
&= \underline{4}
\end{aligned}$$

4 Funzioni di interpretazione semantica

Diamo la semantica del linguaggio in stile *denotazionale* attraverso la definizione di alcune funzioni di interpretazione semantica. Le principali sono:

- Comandi: $\mathcal{S}em_c : \texttt{Com} \to P \to M \to M$
- Espressioni: $\mathcal{S}em_e : \texttt{Exp} \to P \to M \to \texttt{Val}$
- Dichiarazioni: $\mathcal{S}em_d : \texttt{Dec} \to P \to M \to (P, M)$

La definizione delle funzioni avviene per casi rispetto alla struttura sintattica dei vari costrutti. Inoltre la semantica viene fornita in modo *composizionale*: la semantica di un costrutto viene data in termini della semantica delle sue componenti.

4.1 Semantica delle espressioni

Ricordiamo la sintassi delle espressioni.

```
Exp ::= Num | Ide | Exp Aop Exp | Exp Bop Exp | Uop Exp | (Exp)

Aop ::= + | - | * | / | %
Bop ::= > | >= | < | <= | == | != | && | ||
Uop ::= !
```

Nel linguaggio che stiamo considerando non esiste il tipo dei booleani. La costante 0 è interpretata come $false$ e un qualunque altro valore intero diverso da 0 come *true*.

$\texttt{n} \in \texttt{Num}$

$$\mathcal{S}em_e\ \texttt{n}\ \rho\ \mu = val\ \texttt{n}$$

dove si assume data la funzione *val* che, dato $\texttt{n} \in \texttt{Num}$ fornisce il valore intero la cui rappresentazione è n.

$\texttt{x} \in \texttt{Ide}$

$$\mathcal{S}em_e\ \texttt{x}\ \rho\ \mu = \mu(\rho\ \texttt{x})$$

$\texttt{op} \in \texttt{Aop}$

$$\mathcal{S}em_e\ [\texttt{e}_1\ \texttt{op}\ \texttt{e}_2]\ \rho\ \mu = (\mathcal{S}em_{aop}\ \texttt{op})\ (\mathcal{S}em_e\ \texttt{e}_1\ \rho\ \mu)\ (\mathcal{S}em_e\ \texttt{e}_2\ \rho\ \mu)$$

dove si assume definita la funzione di interpretazione semantica $\mathcal{S}em_{aop}$ per gli operatori numerici. Le parentesi quadre usate in $\mathcal{S}em_e\ [\texttt{e}_1\ \texttt{op}\ \texttt{e}_2]\ \rho\ \mu$ non fanno parte della sintassi del linguaggio, ma stanno semplicemente ad evidenziare che l'intera espressione $[\texttt{e}_1\ \texttt{op}\ \texttt{e}_2]$ è il primo argomento di $\mathcal{S}em_c$ in questa regola. Useremo lo stesso accorgimento anche per il resto di queste note, laddove necessario. Infine:

$\texttt{bop} \in \texttt{Bop}$

$$\mathcal{S}em_e\ [\texttt{e}_1\ \texttt{bop}\ \texttt{e}_2]\ \rho\ \mu = (\mathcal{S}em_{bop}\ \texttt{bop})\ (\mathcal{S}em_e\ \texttt{e}_1\ \rho\ \mu)\ (\mathcal{S}em_e\ \texttt{e}_2\ \rho\ \mu)$$

$$\mathcal{S}em_e\ \texttt{!e}\ \rho\ \mu = \neg\ (\mathcal{S}em_e\ \texttt{e}\ \rho\ \mu)$$

Anche in questo caso si assume nota la funzione di interpretazione semantica $\mathcal{S}em_{bop}$ per gli operatori logici e di confronto. Si assume inoltre che la negazione $\neg$ sia definita come segue:

$$\begin{array}{rcll} \neg 0 & = & 1 & \\ \neg n & = & 0 & \quad \texttt{con}\ n \neq 0 \end{array}$$

Veniamo infine all'ultima regola semantica per le espressioni.

$$\mathcal{S}em_e(\texttt{e})\ \rho\ \mu = \mathcal{S}em_e\ \texttt{e}\ \rho\ \mu$$

4.2 Implementazione CAML di $\mathcal{S}em_e$

```
let sembop o=
  match o with
     Add -> let f v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValN (n1+n2) in f |
     Sub -> let f v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValN (n1-n2) in f |
     Mul -> let f v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValN (n1*n2) in f |
     Div -> let f  v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValN (n1/n2) in f |
     Mod -> let f  v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValN (n1 mod n2) in f |
     Gt -> let f  v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValB (n1 > n2) in f |
     Gte -> let f  v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValB (n1 >= n2) in f |
     Lt -> let f  v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValB (n1 < n2) in f |
     Lte -> let f  v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValB (n1 <= n2) in f |
     Eq -> let f  v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValB (n1 = n2) in f |
     Neq -> let f  v1 v2 = match (v1,v2) with
                              (ValN n1,ValN n2) -> ValB (n1 <> n2) in f |
     And -> let f  v1 v2 = match (v1,v2) with
                              (ValB b1,ValB b2) -> ValB (b1 & b2) in f |
     Or -> let f  v1 v2 = match (v1,v2) with
                              (ValB b1,ValB b2) -> ValB (b1 or b2) in f ;;

let semuop o =
  match o with
     Minus -> let f v1 = match v1 with
                              ValN n -> ValN (-n) in f |
     Not -> let f v1 = match v1 with
                              ValB b -> ValB (not b) in f;;


(* SEM_e *)
let rec semexp e (a:amb list) (m:mem list) =
  match e with
    Ide i -> (let (Def l) = search a i in
                 let (Def v) = search m l in v) |
    Num n -> ValN n |
    Bool b -> ValB b |
    BinExp (e1,o,e2) -> sembop o (semexp e1 a m) (semexp e2 a m)|
    UnExp (o,e1) -> semuop o (semexp e1 a m);;
```

Tipizzazione

```
sembop : bop -> val -> val -> val
semuop : uop -> val -> val
semexp : exp -> amb list -> mem list -> val = <fun>
```

4.3 Semantica delle dichiarazioni

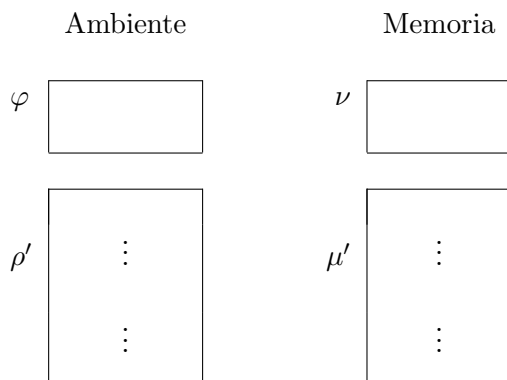
Ricordiamo la sintassi delle dichiarazioni, tralasciando per il momento i puntatori.

```
Dec ::= Type Ide; | Type Ide = Exp;

Type ::= BType

BType ::= int

DecList ::= Dec | Dec DecList
```

L'effetto delle dichiarazioni è di introdurre nuove associazioni nello stato. Intuitivamente, dato uno stato costituito dalle pile ambiente e memoria così rappresentate:

L'effetto della dichiarazione

```
T x;
```

è di *aggiungere* le dovute associazioni per `x` nel frame ambiente e nel frame memoria in cima alle rispettive pile, ovvero in φ e ν. Dobbiamo quindi fare in modo che, per effetto della dichiarazione, nel nuovo stato che si ottiene

(1) il frame ambiente in cima alla pila ambiente sia ottenuto a partire da φ associando ad `x` una *nuova* locazione ℓ, ovvero una locazione tale che $\nu(\ell) = \bot$;

(2) il frame memoria in cima alla pila memoria sia ottenuto a partire da ν associando alla locazione ℓ un valore. Poiché la dichiarazione non specifica tale valore, indicheremo con il simbolo ? un generico valore *imprecisato*[2].

[2] N.B. dato un frame f, il significato di $f(x) = ?$ è che f contiene una associazione per x, ma il valore associato ad x non è noto, mentre il significato di $f(x) = \bot$ è che ad x non è associato alcun valore.

Al fine di poter selezionare la nuova locazione da associare all'identificatore che si sta dichiarando, supponiamo di disporre di una funzione

$$succloc : M \longrightarrow Loc$$

tale che, se $succloc(\mu) = \ell$, allora $\mu(\ell) = \perp$. Formalmente, possiamo allora descrivere la semantica della dichiarazione vista come segue:

$$\mathcal{S}em_d \ [\texttt{T x;}] \ \rho \ \mu = \langle \rho[^{\ell}/_{\texttt{x}}]^{add}, \mu[^{?}/_{\ell}]^{add} \rangle$$

dove $\ell = succloc \ \mu$

L'effetto della dichiarazione

```
T x = E;
```

è del tutto analogo al caso precedente, con la differenza che il valore associato alla nuova locazione è il valore dell'espressione `E` ottenuto mediante la funzione $\mathcal{S}em_e$.

$$\mathcal{S}em_d \ [\texttt{T x = E;}] \ \rho \ \mu = \langle \rho[^{\ell}/_{\texttt{x}}]^{add}, \mu[^{v}/_{\ell}]^{add} \rangle$$

dove $v = \mathcal{S}em_e \ \texttt{E} \ \rho \ \mu$

e $\ell = succloc \ \mu$

Si noti che la semantica di `T x = E;` evidenzia il fatto che l'espressione `E` deve avere un significato (ovvero un valore) nello stato: in particolare, se in `E` compaiono degli identificatori, questi devono essere parte dello stato ed avere in esso un valore.

Veniamo infine alla semantica di una sequenza di dichiarazioni, per la quale viene definita una funzione di interpretazione semantica

$$\mathcal{S}em_{dl} : \texttt{DecList} \rightarrow P \ \rightarrow M \ \rightarrow (P \ , M \ )$$

definita come segue

$$\mathcal{S}em_{dl} \ [\texttt{D DList}] \ \rho \ \mu = \mathcal{S}em_{dl} \ \texttt{DList} \ \rho' \ \mu'$$

dove $\langle \rho', \mu' \rangle = \mathcal{S}em_d \ \texttt{D} \ \rho \ \mu$

$$\mathcal{S}em_{dl} \ \texttt{D} \ \rho \ \mu = \mathcal{S}em_d \ \texttt{D} \ \rho \ \mu$$

Vediamo come esempio la seguente sequenza di dichiarazioni

```
int x; int y = 10;
```

e valutiamone la semantica a partire da uno stato $\langle \rho_0, \mu_0 \rangle = \langle \omega.\Omega, \omega.\Omega \rangle$ in cui le pile ambiente e memoria contengono solo un frame vuoto. Valutiamo dapprima:

$\mathcal{S}em_d$ (`int x;`) $\omega.\Omega\ \omega.\Omega$

$=$ $\{\ \ell_x = succloc\ \omega.\Omega\ \}$

$\langle (\omega.\Omega)[^{\ell_x}/_{\mathtt{x}}]^{add}, (\omega.\Omega)[^?/_{\ell_x}]^{add} \rangle$

$=$ $\{$ definizione di $[\ /\ ]^{add}$ $\}$

$\omega[^{\ell_x}/_{\mathtt{x}}]^{add}.\Omega, \omega[^?/_{\ell_x}]^{add}.\Omega$

$=$

$\langle \rho_1, \mu_1 \rangle$

Abbiamo quindi

$\mathcal{S}em_{dl}$ (`int x; int y=10;`) $\omega.\Omega\ \omega.\Omega$

$=$ $\{$ definizione di $\mathcal{S}em_{dl}$ e dim. precedente $\}$

$\mathcal{S}em_{dl}$ (`int y=10;`) $\rho_1\ \mu_1$

$=$ $\{$ definizione di $\mathcal{S}em_{dl}$ $\}$

$\mathcal{S}em_d$ (`int y=10;`) $\rho_1\ \mu_1$

$=$ $\{\ \mathcal{S}em_e\ \mathtt{10}\ \rho_1\ \mu_1 = 10,\ \ell_y = succloc\ \mu_1\ \}$

$\langle \rho_1[^{\ell_y}/_{\mathtt{y}}]^{add}, \mu_1[^{10}/_{\ell_y}]^{add} \rangle$

$=$ $\{$ definizione di $[\ /\ ]^{add}$ e di $\langle \rho_1, \mu_1 \rangle$ $\}$

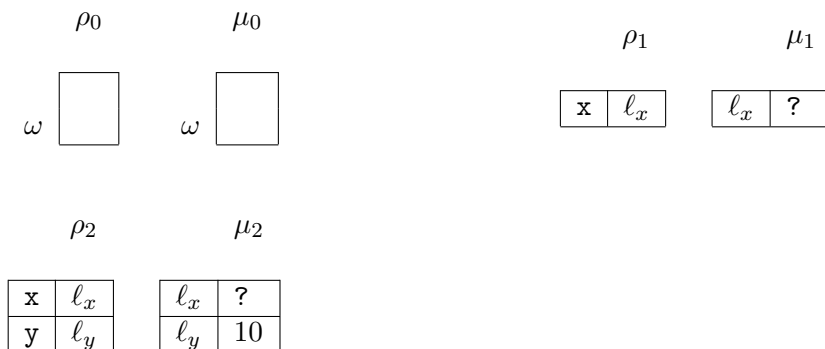
$\langle (\omega[^{\ell_x}/_{\mathtt{x}}]^{add})[^{\ell_y}/_{\mathtt{y}}]^{add}.\Omega, (\omega[^?/_{\ell_x}]^{add})[^{10}/_{\ell_y}]^{add}.\Omega \rangle$

$=$ $\{$ notazione semplificata $\}$

$\langle \omega[^{\ell_x}/_{\mathtt{x}}, ^{\ell_y}/_{\mathtt{y}}]^{add}.\Omega, \omega[^?/_{\ell_x}, ^{10}/_{\ell_y}]^{add}.\Omega \rangle$

$=$

$\langle \rho_2, \mu_2 \rangle$

Una rappresentazione grafica di quanto appena mostrato è la seguente:

4.4 Implementazione CAML di $\mathcal{S}em_d$ e $\mathcal{S}em_{dl}$

```
let succloc (m:mem list) =
   let rec findunused m n = match search m n with
                           Bottom -> (n:mloc) |
                           _       -> findunused m (n+1)
   in findunused m 0;;

(* SINGOLA DICHIARAZIONE *)
let semd d (a:amb list) (m : mem list) = match d with
      Var x -> ((add_stack a  x (Def (succloc m)), add_stack m (succloc m) (Def Unknown))
                            : (amb list * mem list) )   |
      Var_init(x,e) -> let v = (semexp e a m) in
                    (add_stack a  x (Def (succloc m)), add_stack m (succloc m) (Def v));;

(* SEQUENZA DI DICHIARAZIONI *)
let rec semdl dl (a:amb list) (m : mem list) = match dl with
      [] -> (a, m) |
      d::ds -> let a',m'=semd d a m in semdl ds a' m';;
```

Tipizzazione

```
succloc : mem list -> mloc
semd : dec -> amb list -> mem list -> amb list * mem list
semdl : dec list -> amb list -> mem list -> amb list * mem list
```

4.5 Semantica dei comandi

Ricordiamo la sintassi dei comandi.

```
Com ::= Ide = Exp; | if (Exp) Com else Com | if (Exp) Com | while (Exp) Com | Block

Block ::= {DecComList}

DecComList ::= DecList ComList | ComList

DecList ::= Dec | Dec DecList
ComList ::= Com | Com ComList
```

I comandi hanno l'effetto di modificare lo stato, nella sua parte modificabile, ovvero la memoria. Dunque

$$\mathcal{S}em_c : \texttt{Com} \to P \ \to M \ \to M$$

Come già fatto per le espressioni e le dichiarazioni, diamo le regole per $\mathcal{S}em_c$ in modo *guidato dalla sintassi*, ovvero definiamo $\mathcal{S}em_c$ per casi in base alle possibili alternative sintattiche.

Assegnamento

Lo scopo di un assegnamento `id = E` è di modificare il valore associato nello stato all'identificatore `id`, facendo in modo che diventi il valore dell'espressione `E` nello stato corrente.
Ricordiamo che, in uno stato $\langle \rho, \mu \rangle$, il valore associato ad un identificatore `id` è il valore associato nella memoria alla locazione $\rho(\texttt{id})$ e che quest'ultima è la locazione associata ad `id` nel "primo" frame della pila ambiente per cui $\rho(\texttt{id}) \neq \perp$.

$$\mathcal{S}em_c\ [\texttt{id = E}]\ \rho\ \mu = \mu[^v/_{\rho(\texttt{id})}]^{mod}$$

dove $v = \mathcal{S}em_e\ \texttt{E}\ \rho\ \mu$

Condizionale

Ricordiamo che, nella semantica che stiamo definendo, i valori di verità sono modellati da 0 ($false$) e da un valore intero diverso da 0 ($true$).

$$\mathcal{S}em_c\ [\texttt{if (E) C1 else C2}]\ \rho\ \mu = \mathcal{S}em_c\ (\texttt{C1})\ \rho\ \mu$$

se $\mathcal{S}em_e\ \texttt{E}\ \rho\ \mu \neq 0$

$$\mathcal{S}em_c\ [\texttt{if (E) C1 else C2}]\ \rho\ \mu = \mathcal{S}em_c\ (\texttt{C2})\ \rho\ \mu$$

se $\mathcal{S}em_e\ \texttt{E}\ \rho\ \mu = 0$

La semantica del condizionale privo di ramo "else" è analoga.

$$\mathcal{S}em_c\ [\texttt{if (E) C}]\ \rho\ \mu = \mathcal{S}em_c\ (\texttt{C1})\ \rho\ \mu$$

se $\mathcal{S}em_e\ \texttt{E}\ \rho\ \mu \neq 0$

$$\mathcal{S}em_c\ [\texttt{if (E) C}]\ \rho\ \mu = \mu$$

se $\mathcal{S}em_e\ \texttt{E}\ \rho\ \mu = 0$

Sequenza di comandi e blocchi

Come già fatto per le dichiarazioni, introduciamo una nuova funzione di interpretazione semantica

$$\mathcal{S}em_{cl} : \texttt{ComList} \rightarrow P\ \rightarrow M\ \rightarrow M$$

definita come segue per casi sulla struttura sintattica di `ComList`.

$$\mathcal{S}em_{cl}\ [\texttt{C CList}]\ \rho\ \mu = \mathcal{S}em_{cl}\ \texttt{CList}\ \rho\ \mu'$$

dove $\mu' = \mathcal{S}em_c\ \texttt{C}\ \rho\ \mu$

$$\mathcal{S}em_{cl}\ \texttt{C}\ \rho\ \mu = \mathcal{S}em_c\ \texttt{C}\ \rho\ \mu$$

Dunque, nella sequenza `C CList`, andiamo a valutare `CList` in uno stato in cui la componente modificabile (la memoria) è quella risultante dalla valutazione (esecuzione) del primo comando (`C`) dell'intera sequenza. Vediamo un esempio

Vediamo come esempio la seguente lista di comandi

```
x = 25; y = x + 1;
```

e valutiamone la semantica a partire dallo stato $\langle \rho_0, \mu_0 \rangle$ rappresentato dalla seguente figura.

ρ_0

x	ℓ_x
y	ℓ_y

μ_0

ℓ_x	100
ℓ_y	-5

Valutiamo dapprima

$\mathcal{S}em_c$ (`x = 25;`) $\rho_0\ \mu_0$

$=$ $\{\ 25 = \mathcal{S}em_e\ 25\ \rho_0\ \mu_0\ \}$

$\mu_0[^{25}/_{\ell_x}]^{mod}$

$=$

μ_1

Lo stato $\langle \rho_0, \mu_1 \rangle$ in cui andremo a valutare il secondo comando della sequenza è quello così rappresentato

ρ_0

x	ℓ_x
y	ℓ_y

μ_1

ℓ_x	25
ℓ_y	-5

Abbiamo quindi

$\mathcal{S}em_{cl}$ (`x = 25; y = x+1;`) $\rho_0\ \mu_0$

$=$ $\{$ definizione di $\mathcal{S}em_{cl}$ e dim. precedente $\}$

$\mathcal{S}em_{cl}$ (`y = x+1;`) $\rho_0\ \mu_1$

$=$ $\{$ definizione di $\mathcal{S}em_{cl}$ $\}$

$\mathcal{S}em_c$ (`y = x+1;`) $\rho_0\ \mu_1$

$=$ $\{$ $\mathcal{S}em_e$ $\mathtt{x}+\mathtt{1}\ \rho_0\ \mu_1 = 26$ $\}$

$\mu_1[^{26}/_{\ell_y}]^{mod}$

$=$

μ_2

La memoria μ_2 risultante dalla valutazione della sequenza è dunque la seguente

μ_2

ℓ_x	25
ℓ_y	26

Blocchi

Un blocco, nella sua forma più generale, è costituito da una sequenza di dichiarazioni e da una sequenza di comandi. La visibilità degli identificatori dichiarati in un blocco (gli identificatori *locali* del blocco) è il blocco stesso. Inoltre, se un identificatore `x` già presente nello stato viene (ri-)dichiarato in un blocco, la sua visibilità all'interno del blocco "nasconde" la visibilità degli identificatori con lo stesso nome già presenti nello stato . Dal punto di vista semantico, la corretta gestione di tali situazioni, è garantita utilizzando opportunamente la struttura a pila dello stato. Infatti, la valutazione della lista di comandi di un blocco avviene in uno stato in cui la pila ambiente e la pila memoria sono state arricchite di un nuovo frame che mantiene le opportune nuove associazioni per gli identificatori e le corrispondenti locazioni dichiarate all'interno del blocco. Formalmente:

$$\mathcal{S}em_c\ \{\mathtt{Dlist\ CList}\}\ \rho\ \mu = \mu''$$

dove $\mathcal{S}em_{dl}$ $\mathtt{Dlist}\ \omega.\rho\ \omega.\mu = \langle\varphi.\rho, \nu.\mu\rangle$

e $\mathcal{S}em_{cl}$ $\mathtt{Clist}\ \varphi.\rho\ \nu.\mu = \nu'.\mu''$

Osservazioni

1) le dichiarazioni del blocco vanno a "popolare" i frame (inizialmente vuoti) che vengono impilati sulle pile ambiente memoria
2) nello stato risultante viene valutata la sequenza di programmi del blocco che restituisce una memoria del tipo $\nu'.\mu''$ dove ν' è un frame che contiene i valori associati agli identificatori locali del blocco e μ'' contiene i valori associati agli identificatori non locali
3) nella memoria μ'' risultante dalla valutazione dell'intero blocco permangono solo le modifiche apportate dai comandi del blocco alle sole associazioni degli identificatori non locali
4) all'uscita dal blocco non vi è più modo di accedere agli identificatori locali al blocco stesso.

Il caso di blocco privo di dichiarazioni locali viene trattato allo stesso modo per uniformità.

$$\mathcal{S}em_c\ \{\texttt{CList}\}\ \rho\ \mu = \mu'$$

dove $\mathcal{S}em_{cl}\ \texttt{Clist}\ \omega.\rho\ \omega.\mu = \omega.\mu'$

Si noti come la regola metta in evidenza il fatto che, nella memoria risultante dalla valutazione della lista di comandi, il frame in cima alla pila rimane vuoto.

Iterazione

Dato un comando iterativo `while (E) C` chiamiamo *guardia* l'espressione booleana `E` e *corpo* il comando `C`. La semantica intuitiva corrisponde alla valutazione di una sequenza del tipo

$$\underbrace{\texttt{C C}\ldots\texttt{C}\ldots}$$

dove il corpo `C` viene valutato fino al raggiungimento di uno stato in cui la guardia `E` è falsa. La sequenza può essere la sequenza vuota nel caso in cui la valutazione della guardia sia falsa al momento della valutazione dell'intero comando. Formalmente:

$$\mathcal{S}em_c\ [\texttt{while (E) C}]\ \rho\ \mu = \mu$$

se $\mathcal{S}em_e\ \texttt{E}\ \rho\ \mu = 0$

$$\mathcal{S}em_c\ [\texttt{while (E) C}]\ \rho\ \mu = \mu''$$

se $\mathcal{S}em_e\ \texttt{E}\ \rho\ \mu \neq 0$

dove $\mathcal{S}em_c\ \texttt{C}\ \rho\ \mu = \mu'$

e $\mathcal{S}em_c\ [\texttt{while (E) C}]\ \rho\ \mu' = \mu''$

4.6 Implementazione CAML di $\mathcal{S}em_c$ e $\mathcal{S}em_{cl}$

```
(* SEMANTICA DEI COMANDI *)
let rec semc c (a:amb list) (m:mem list) = match c with
     Assign(x,e) -> let v = (semexp e a m) and (Def l) = (search a x) in
                   (update_stack m l (Def v): mem list) |
     If_then_else(e,c1,c2) -> (match semexp e a m with
                     ValB true -> semc c1 a m |
                     ValB false -> semc c2 a m ) |
     If_then(e,c1) -> (match semexp e a m with
                     ValB true -> semc c1 a m |
                     ValB false -> m ) |
     While(e,c1) -> (match semexp e a m with
                     ValB false -> m |
                     ValB true -> let m' = semc c1 a m in
                                     semc (While(e,c1)) a m') |
     Block(dl,cl) -> let (a', m') = semdl dl (omega::a) (omega::m)
                 in let (fm :: m'') = semcl cl a' m' in m''
and

semcl cl a m = match cl with
     [] -> m |
     c::cs -> semcl cs a (semc c a m) ;;
```

Tipizzazione

```
semc : com -> amb list -> mem list -> mem list = <fun>
semcl : com list -> amb list -> mem list -> mem list
```

4.7 Programmi

Ricordiamo la sintassi dei programmi nel nostro linguaggio

```
Prog ::= Block
```

Dunque un programma è un blocco del tipo

```
{DL  CL}
```

La funzione di interpretazione semantica per i programmi

$$\mathcal{S}em_{prog} : \texttt{Prog} \rightarrow M$$

è semplicemente la semantica del blocco che costituisce il programma a partire da uno stato *vuoto*.

$$\mathcal{S}em_{prog}\ \{\texttt{DL CL}\} = \mathcal{S}em_c\ \{\texttt{DL CL}\}\ \Omega\ \Omega$$

Se osserviamo attentamente, otteniamo

$\mathcal{S}em_{prog}\{$`DL CL`$\}$

$=$ $\{$ definizione di $\mathcal{S}em_{prog}$ $\}$

$\mathcal{S}em_c$ $\{$`DL CL`$\}$ Ω Ω

$=$ $\{$ def. di $\mathcal{S}em_c$, con $\mathcal{S}em_{dl}$ `Dl` $\omega.\Omega\ \omega.\Omega = \langle\varphi.\Omega, \nu.\Omega\rangle$ e $\mathcal{S}em_{cl}$ `Clist` $\varphi.\Omega\ \nu.\Omega = \nu'.\Omega$ $\}$

Ω

Dunque il programma viene valutato a partire da uno stato vuoto e restituisce una memoria vuota! Nei linguaggi di programmazione reali, vengono utilizzati i comandi di *input/output* per ottenere dall'esterno i dati iniziali (*input*) e per comunicare all'esterno i risultati del calcolo (*output*).

4.8 Implementazione CAML di $\mathcal{S}em_{prog}$

```
let semprog p = match p with
      Prog(dl,cl) ->  semc (Block(dl,cl)) [] [];;
```

Tipizzazione

```
semprog : prog -> mem list
```